Not actually crossing the airtight hatchway: Harmless out-of-bounds read that is never disclosed

devblogs.microsoft.com/oldnewthing/20191204-00

December 4, 2019

Raymond Chen

A security vulnerability report arrived that went something like this:

> By passing a specifically malformed payload, an attacker can trigger an out-of-bounds read. By this means, a remote attacker can cause the disclosure of sensitive information. An attacker can combine this with other vulnerabilities to achieve remote code execution.

The finder also included some reverse-compiled output[1] highlighting the point at which the out-of-bounds read occurred.

Anyway, it appears that the out-of-bounds read was discovered by using a memory debugging tool that does strict validations of every memory access. But consumers in the wild don't run programs in such an environment.

When run on an actual consumer machine, the program uses the standard operating system heap manager, and the standard operating system heap manager does things like pad allocations to maintain alignment. Those extra bytes are technically off-limits, but they will always be there.

In this case, what happens is that the code allocates a block of memory, then reads past the end of it by a tiny amount, well within the heap padding, so it's reading uninitialized heap memory. No denial of service is possible here because the heap padding saves you.

The next thing the code does is validate that the buffer is valid. This validation fails because the memory block is too small, and the operation fails. The value read from the uninitialized heap memory is not returned, so it is never disclosed to anybody.

Here's a sketch. Assume that the `checked_*` functions reject the request if the operation fails.

```
struct ITEMSLIST
{
  uint32_t itemCount;
  ITEM items[ANYSIZE_ARRAY];
};

auto list = (ITEMSLIST*)checked_malloc(byteCount);
checked_read(list, byteCount);

auto requiredSize =
    checked_add(offsetof(ITEMSLIST, items) +
                checked_mult(sizeof(ITEM), header->itemCount));
checked_require(byteCount >= requiredSize);

... do stuff with the items ...
```

If the `byteCount` is less than `sizeof(uint32_t)`, then the code under-allocates the `list` and tries to read the `itemCount` from it. Oh no, we are at risk of disclosing heap memory!

But then the code checks that the header size is large enough to hold the specified number of items, and seeing as the header size is not even large enough to hold the header, it certainly isn't large enough to hold any items. So the request is rejected.

Note that the invalid `itemCount` never leaves the function. The value of `itemCount` is heap garbage, but whatever value it has will always fail the `byteCount >= requiredSize` test (assuming it manages to pass the `checked_ mult` test), so the call will always be rejected. And the rogue value of `itemCount` is not exposed, so whatever garbage value happened to be there never escapes. What happens in parameter validation stays in parameter validation.

The finder jumped the gun: They found an out-of-bounds read but didn't study it to see whether it was exploitable. They immediately concluded that there was information disclosure, and then tacked on a remote code execution for good measure.

What they found is a defect, but it has no security implications. It's just a bug.

When informed that the issue as not exploitable and therefore has no security implications, they went ahead and issued a security bulletin anyway.

Six months later, the same organization found the same issue in a different component. We again told them that it was not exploitable and therefore has no security implications. The second time, they withdrew their plans to issue a bulletin.

So I'm not sure what changed over there, but at least they stopped issuing bogus bulletins for this category of issue.

Bonus chatter: This category of false alarm is quite common. People use various analysis tools to identify issues and immediately file a report without evaluating whether the issue actually is a vulnerability. They subscribe to the shotgun approach: File tons of potential issues, and let Microsoft figure out which ones are valid. Why do the extra work if you can externalize it!

[1] The reverse-compiled output has meaningless variable names like `v1`, `v2` and `v3`, and object member accesses are expressed in the form `(int*)((BYTE*)v40 + 0x20)`.

A note to people who send reverse-compiled output: Please include the original assembly language, and annotate that. Otherwise, we have to take your reverse-compiled output and try to re-compile it to assembly language in a way that matches the actual binary, and then re-reverse-compile it back to the original source code. These steps can be quite complicated because of compiler optimizations. (Also because people often fail to provide enough build number information to let us identify exactly which binary you are reverse-compiling, forcing us to keep trying all the different patched versions of the binary until we find a match or become exhausted.)

If you're using IDA Pro's Hex-Rays decompiler, you can right-click and say "Copy comments to disassembly." That will take your comments in the reverse-compiled code and apply them to the corresponding lines of assembly.

Raymond Chen

Follow